Relatório: IPC, Threads e Paralelismo

Gabriel Bartolomeu da Silva[1] e Vinícius Andriani Mazera[2]

Universidade do Vale do Itajaí – UNIVALI
Ciência da Computação

1. Introdução

Este relatório apresenta um código que faz tarefas simples como INSERT, DELETE, SELECT e UPDATE em um banco de dados simulado, utilizando e dando ênfase nos conceitos de IPC, threads, concorrência e paralelismo trabalhados em sala.

2. Explicação do Código

2.1 Trechos do Código

```
int main()
{
    int fd = shm_open("/bd_ipc", O_RDWR, 0666);
    IPCData* ipc_data = (IPCData*)mmap(nullptr, sizeof(IPCData), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Acesso à memória compartilhada e mapeamento dela.

```
while (true)
{
    std::string comando;
    std::getline(std::cin, comando);

    if (comando == "exit") break;

    strncpy(ipc_data->comando, comando.c_str(), sizeof(ipc_data->comando));

    while (strlen(ipc_data->resposta) == 0) {
        usleep(10000);
    }

    std::cout << "Resposta: " << ipc_data->resposta << std::endl;
    memset(ipc_data->resposta, 0, sizeof(ipc_data->resposta));

}

munmap(ipc_data, sizeof(IPCData));
return 0;
```

Loop até que seja requisitado “exit” no cliente, envia o comando para o servidor, aguarda resposta e em seguida, a envia.

```
Registro banco[MAX_REGISTROS];
int total_registros = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Declara o banco simulado para registros e inicializa o mutex.

```
int main()
{
    int fd = shm_open("/bd_ipc", O_CREAT | O_RDWR, 0666);
    ftruncate(fd, sizeof(IPCData));
    IPCData* ipc_data = (IPCData*)mmap(nullptr, sizeof(IPCData), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    pthread_t thread1, thread2;
    pthread_create(&thread1, nullptr, thread_handler, ipc_data);
    pthread_create(&thread2, nullptr, thread_handler, ipc_data);

    std::cout << "Servidor iniciado. Aguardando comandos..." << std::endl;

    pthread_join(thread1, nullptr);
    pthread_join(thread2, nullptr);

    munmap(ipc_data, sizeof(IPCData));
    shm_unlink("/bd_ipc");
    return 0;
}
```

Acesso à memória compartilhada e mapeamento dela, cria pool de threads e faz suas junções, em seguida faz a limpeza.

```
//handler de requisições
void* thread_handler(void* arg)
{
    IPCData* ipc_data = (IPCData*)arg;

    while (true)
    {
        if (strlen(ipc_data->comando) > 0)
        {
            processar_comando(ipc_data->comando, ipc_data->resposta);
            memset(ipc_data->comando, 0, sizeof(ipc_data->comando));
        }
        usleep(100000);
    }
    return nullptr;
}
```

Thread para tratar as requisições do cliente, loop que chama o processamento dos comandos e os limpa.

```
void processar_comando(const char* comando, char* resposta)
{
    pthread_mutex_lock(&mutex);

    //ex: INSERT id=1 nome=Vinicius
    if (strstr(comando, "INSERT") != nullptr)
    {
        int id;
        char nome[TAM_NOME];
        sscanf(comando, "INSERT id=%d nome=%49s", &id, nome);

        bool id_existe = false;
        for (int i = 0; i < total_registros; i++)
        {
            if (banco[i].id == id)
            {
                id_existe = true;
                break;
            }
        }
        if (id_existe)
        {
            strcpy(resposta, "ERRO: id já existe :c");
        }
        else if (total_registros < MAX_REGISTROS)
        {
            banco[total_registros].id = id;
            strcpy(banco[total_registros].nome, nome);
            total_registros++;
            strcpy(resposta, "INSERT OK");
        } else
```

Processamento do comando, trava o mutex ao iniciar, verifica se o comando for INSERT, se o banco está cheio e se o id já está presente no banco, então insere o cadastro.

```
else if (strstr(comando, "DELETE") != nullptr)
{
    int id;
    sscanf(comando, "DELETE id=%d", &id);

    bool id_inexiste = true;
    for (int i = 0; i < total_registros; i++)
    {
        if (banco[i].id == id)
        {
            id_inexiste = false;
            break;
        }
    }
    if (id_inexiste)
    {
        strcpy(resposta, "ERRO: id não encontrado :c");
    }
    else
    {
        for (int i = 0; i < total_registros; i++)
        {
            if (banco[i].id == id)
            {
                banco[i] = banco[total_registros - 1];
                total_registros--;
                strcpy(resposta, "DELETE OK :D");
                break;
            }
        }
```

Caso o comando seja DELETE, verifica se há cadastro com o id requisitado na deleção, então o deleta do banco.

```
else if (strstr(comando, "SELECT") != nullptr)
{
    int id;

    sscanf(comando, "SELECT id=%d", &id);

    bool id_inexiste = true;
    for (int i = 0; i < total_registros; i++)
    {
        if (banco[i].id == id)
        {
            id_inexiste = false;
            break;
        }
    }
    if (id_inexiste)
    {
        strcpy(resposta, "ERRO: id não encontrado :c");
    }
    else
    {
        snprintf(resposta, TAM_RESPOSTA, "REGISTRO ENCONTRADO:\n Nome: %s", banco[id-1].nome);
    }
}
```

Para comando SELECT, envia um id, se estiver presente no banco, recebe como resposta o nome correspondente ao id.

```
else if (strstr(comando, "UPDATE") != nullptr)
{
    int id;
    char novoNome[TAM_NOME];
    sscanf(comando, "UPDATE id=%d NOME=%s", &id, novoNome);

    bool id_inexiste = true;
    for (int i = 0; i < total_registros; i++)
    {
        if (banco[i].id == id)
        {
            id_inexiste = false;
            break;
        }
    }
    if (id_inexiste)
    {
        strcpy(resposta, "ERRO: id não encontrado :c");
    }
    else
    {
        int tamNom = strlen(novoNome);
        memcpy(banco[id].nome, novoNome, tamNom);
        snprintf(resposta, TAM_RESPOSTA, "REGISTRO ATUALIZADO:\n Nome: %s", banco[id].nome);
    }
```

Para comando UPDATE, é informado na requisição o id da tupla a ser modificada e o novo nome após modificação, se o id existe no banco, é feita a troca de nome.

```
else
{
    strcpy(resposta, "COMANDO DESCONHECIDO/INCORRETO");
}

pthread_mutex_unlock(&mutex);
```

Caso não seja nenhum dos comandos mencionados anteriormente, notifica que o comando é desconhecido ou está incorreto. Como fim, libera o mutex.

2.2 Execução do Código

No terminal do codespace, segue o passo a passo:

- compilar:

bash

copy

make clean && make

- iniciar o Servidor (em um terminal):

bash

copy

./server

- iniciar o Cliente (em outro terminal):

bash

copy

./cliente

3. Conclusão

O código foi uma ferramenta útil para entender melhor a natureza e possíveis dificuldades dos conceitos de threads, concorrência, paralelismo e IPC, principalmente. Apesar da finalidade do código ser o gerenciamento de um banco de dados simulado muito simples, houve dificuldades pelo uso de IPC em threads, onde há inconsistência nas respostas do servidor.